PHAR LAP 386 DEBUG REFERENCE MANUAL

386 DOS-EXTENDER SDK

# 386 | DEBUG Reference Manual

Phar Lap Software, Inc. 60 Aberdeen Avenue, Cambridge, MA. 02138 (617) 661-1510, FAX (617) 876-2972 dox@pharlap.com tech-support@pharlap.com Copyright 1986-91 by Phar Lap Software, Inc.

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of Phar Lap Software, Inc. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013.

First edition: September 1987. Second edition: January 1988. Third edition: January 1991.

Phar Lap® is a registered trademark of Phar Lap Software, Inc. 386 | DOS-Extender™ is a trademark of Phar Lap Software, Inc.

Cyrix® and EMC87® are registered trademarks of Cyrix Corporation.

Intel® is a registered trademark of Intel Corporation.

386<sup>™</sup> is a trademark of Intel Corporation.

Macintosh® is a registered trademark of Apple Corporation.

MacDraw® is a registered trademark of CLARIS Corporation.

MetaWare High C® and MetaWare Professional Pascal® are registered trademarks of MetaWare Incorporated.

MS® and MS-DOS® are registered trademarks of Microsoft Corporation.

NDP C-386<sup>™</sup> and NDP Fortran-386<sup>™</sup> are trademarks of Microway, Inc.

WATCOM™ is a trademark of WATCOM Products Inc.

Weitek® is a registered trademark of Weitek Corporation.



# **Table of Contents**

Preface		ix
Chapter 1	Introducing 386   DEBUG	1
Chapter 2	Using 386   DEBUG	3
2.1	Command Line Syntax	3
2.2	Command Line Switches	4
2.2.1	I/O Redirection Switches	
2.2.2	Case Sensitivity Switches	
2.2.3	Default Pointer Size Switches	
2.2.4	Loading a Different Symbol Table Switch	
2.2.5	386 DOS-Extender Switches	
2.3	Starting 386   DEBUG	9
2.4	Command Formats	10
2.4.1	Command Operands	
2.4.2	Symbols	
2.4.3	Expressions	
2.4.4	Addresses	
2.4.5	Ranges	
2.4.6	Expression Operators	
2.5	Loading Alternate Symbol Tables	19
2.6	Controlling 386   DEBUG Display Output	20
2.7	Input and Output Redirection	20
2.7.1	I/O Redirection to a Communications Port	
2.7.2	I/O Redirection to Disk Files	

2.8	Regaining Control	23
2.8.1	Program Breakpoints and Instruction Tracing	
2.8.2	Data Watchpoints	
2.8.3	Processor Exceptions	
2.8.4	Program Termination	
2.8.5	CTRL/C	
2.9	Protected Mode versus Real Mode Debugging	26
Chapter 3	386   DEBUG Command Reference	29
3.1	BC, BD, BE, BL, and BP Commands — Breakpoint	
	Control	29
3.2	C Command — Compare Memory	31
3.3	COM1, COM2, and CON Commands — I/O	
	Redirection	32
3.4	D, DA, DB, DD, and DW Commands — Display	
	Memory	33
3.5	DF, DS, DQ, and DT Commands — Display	
	Memory as Floating Point	35
3.6	DG, DI, and DL Commands — Display System	
	Tables	36
3.7	DTSS, DTSS16, and DTSS32 Commands — Display	
0.0	Task State Segment	38
3.8	E, EB, ED, and EW Commands — Enter (Change)	
0.0	Memory	39
3.9	F Command — Fill Memory	40
3.10	G Command — Go	41
3.11	H Command — Perform Hex Arithmetic	42
3.12	I Command — Input from Port	43
3.13	K and KA Commands — Stack Trace	43
3.14	M Command — Move Memory	45
3.15	MC, ML, MM, MS, and M? Commands — Select	
	Memory Model	46
3.16	O Command — Output to Port	47
3.17	P and PQ Commands — Procedure Trace	
	(Single Step Across CALL and INT Instructions)	47
3.18	PI and PL Commands — Display Paging	
	Information	48
3.19	PROT Command — Assume Protected Mode	49
3.20	Q Command — Quit	50
3.21	R Command — Display or Change Registers	50
3.22	R67, R67D, R67S, and R87 Commands — Display	
	Floating Point Coprocessor Registers	52

3.23	REAL Command — Assume Real Mode	
	Addressing	54
3.24	RX Command — Display Extended Registers	54
3.25	S Command — Search Memory for String	55
3.26	T and TQ Commands — Trace (Single-Step)	56
3.27	U, U16, U32 Commands — Unassemble	
	(Disassemble) a Block of Memory	57
3.28	WC, WD, WE, WL, and WP Commands —	
	Watchpoint Control	60
3.29	X Command — Display Symbol Table	62
3.30	XR Command — Relocate Symbol Table	62
3.31	XS Command — Display Segment Names	63
3.32	XW Command — Locate Symbol	64
3.33	? Command — On Line Help	65
Appendix A	386   DEBUG Command Line	67
Appendix B	Error Messages	71
B.1	Program Load Errors	71
B.2	Command Errors	73



## Preface

This manual, 386 | DEBUG Reference Manual, is the documentation for the Phar Lap symbolic debugger, 386 | DEBUG. It is written for MS-DOS programmers who are already familiar with developing and debugging real mode programs for the Intel 80x86 family of microprocessors, and is not intended as a teaching device; it is for reference and informational purposes only. It is organized in a way that we hope will make it easy for you to find the information you need, with the following chapters and appendices:

Chapter 1: Introducing 386 | DEBUG.

Chapter 2: Using 386 | DEBUG. This chapter includes information on

the command line syntax, switches, formats, and

operands; input and output; control of 386 | DEBUG; and

protected-mode versus real-mode debugging.

Chapter 3: 386 | DEBUG Command Reference. This chapter provides

an alphabetic listing of all 386 | DEBUG commands, their syntax, and a description of their uses and parameters.

Appendix A: 386 | DEBUG Command Line. This appendix provides a

quick reference of the command line options, with a one-

line summary of each command. It also provides a

summary of expression formats.

Appendix B: Error Messages. This appendix lists errors which can

occur in 386 | DEBUG, their causes, and how to correct

them.

If, after you read this book, you find that you have suggestions, improvements, or comments, please call us, write us or send us mail at:

Phar Lap Software, Inc. 60 Aberdeen Ave., Cambridge, MA 02138 (617)661-1510, FAX (617)876-2972 dox@pharlap.com tech-support@pharlap.com

#### **Manual Conventions**

This manual relies on certain conventions to convey certain types of information. On the following pages, these are the conventions.

Courier	indicates command line, switch syntax and examples; this typeface was chosen for its close resemblance to screen
	display and to differentiate actual command lines from
	documentation.
italics	indicate items that must have a user-entered name or
	value entered in place of the items in italics. square brackets indicate that one or more of the enclosed
	items may be chosen. It is valid not to choose any of the
	items in the square brackets.

Commands with address range operands distinguish them as follows:

range If both the start and end addresses must be specified; address | range If the start address is required, but the end address is optional;

[range] If both the start and end addresses are optional.



This manual was produced on a Macintosh IIcx, using MS-Word and MacDraw II. The examples illustrated in this manual were developed with 386 | DEBUG, version 3.0.



## Introducing 386 | DEBUG

386 | DEBUG is a symbolic machine language debugger for the Intel 80386 microprocessor family. It is used to debug programs written to execute in either the protected mode or the real mode of the 80386.

386 | DEBUG provides the same executing environment for protected mode programs as that provided by 386 | DOS-Extender. For real mode programs, 386 | DEBUG uses the standard MS-DOS environment.

386 | DEBUG commands have the same names and general syntax as those found in the SYMDEB utility supplied with MS-DOS. For both protected mode and real mode programs, 386 | DEBUG allows you to display and modify all 80386 registers, and supports disassembly of all 80386 instructions. Additional commands will display, change, move, and compare memory, set breakpoints and watchpoints in a program, and single-step instructions.

A source code debugger, 386 | SRCBug, is available as a separate product from Phar Lap Software.



## Using 386 | DEBUG

This chapter includes information on the basic command line syntax and detailed information on the command line switches, including any arguments each switch takes, in sections 2.1 and 2.2. Section 2.3 provides the information you need to know to start 386 | DEBUG running. Section 2.4 contains a description of command formats, operands, symbols, expressions, addresses, ranges, and expression operators. Section 2.5 documents how to load an alternate symbol table. Sections 2.6 and 2.7 describe display output, and input and output redirection. Section 2.8 discusses when 386 | DEBUG regains control from the application program, and section 2.9 discusses protected-mode versus real-mode debugging.

## 2.1 Command Line Syntax

The debugger is distributed as two files, 386DEBUG.EXE and 386DEBUG.EXP. Copy both files to the same directory on your hard disk.

The command line to run the debugger is 386DEBUG, followed by command line switches, then the name of the program to be debugged and any arguments to be passed to the application program:

386debug [debugswitches] programname [arguments]

The file RUN386.EXE (the 386 | DOS Extender) is required to be in one of the directories specified in the DOS environment variable "PATH". 386 | DEBUG will automatically start 386 | DOS—Extender and then load the program to be debugged. Because 386 | DEBUG searches the execution path for RUN386 just as DOS does, the same version of 386 | DOS—Extender is always used when debugging as when you run your program directly by typing:

run386 programname [arguments]

Some example debugger command lines are:

386debug myprog 386debug -maxreal 200h myprog 386debug pecho This is a protected mode program

Command line switches are used to override the default operation of 386 | DEBUG. The name of the program to be debugged can be supplied in full, or it can be given without a file extension. If no file extension is given, 386 | DEBUG first looks for a file with the extension .EXP (the default file name extension for a protected mode program). If no .EXP file exists, 386 | DEBUG then looks for a file with the extension .EXE (the default file name extension for a real mode program or a bound protected mode program).

386 | DEBUG supports debugging of three program file formats:

- EXP format, the default file format produced by 386ILINK for protected mode programs
- EXE format, the default file format produced by 386ILINK for real mode programs
- REX format, an alternative protected mode program format produced by 386ILINK.

In addition, 386 | DEBUG can be used to debug protected mode programs that are bound with the Phar Lap 386 | DOS-Extender. Please see the 386 | LINK Reference Manual for a complete description of program file formats.

## 2.2 Command Line Switches

Command line switches change the default operation of 386 | DEBUG. By default, 386 | DEBUG:

- Assumes that a small memory model program is being debugged
- Ignores case when looking up program symbols
- Reads debugger command input from the system keyboard, and displays debugger output on the system screen.

Command line switches begin with a minus sign (-) character followed by the name of the switch. There are two forms of each switch name: a long form and a short form. Any argument to the switch must immediately follow the switch name, with a space as a separator. If conflicting switches are given on a command line, the last (rightmost) switch takes precedence.

Some 386 | DEBUG switches take a number as an argument. By default, the number is considered to be a decimal (base 10) number.

Hexadecimal (base 16) numbers may be specified by appending the character "h" or "H" to the number.

### 2.2.1 I/O Redirection Switches

The -COM1, -COM2, and -BAUD switches redirect 386 | DEBUG input and output from the system keyboard and display to a terminal connected to communications port 1 or 2 (on the back of the PC). This is useful when debugging application programs that make extensive use of the system display, since debugger output will not interfere with the application's display setup.

The -COM1 switch redirects debugger I/O to communications port 1. The -COM2 switch redirects I/O to communications port 2. If either of these switches is used without the -BAUD switch, the operations mode of the port is not modified. The DOS MODE command can be used to modify the operations mode of the port.

The -BAUD switch, in conjunction with -COM1 or -COM2, sets the port operations mode. -BAUD takes a single argument, which is the baud rate for the port.

Permissible values are:

110, 150, 300, 600, 1200, 2400, 4800, and 9600.

When the -BAUD switch is used, the rest of the port parameters are automatically set to 8 data bits, one stop bit, and no parity. Use of the

-BAUD switch thus overrides any previous MODE command entered from DOS.

#### Syntax:

- -COM1
- -COM2
- -BAUD rate

#### **Short Form:**

- -COM1
- -COM2
- -BAUD rate

#### Example:

386debug -com1 hello 386debug -com2 -baud 9600 hello

## 2.2.2 Case Sensitivity Switches

The -ONECASE and -TWOCASE switches select the case sensitivity of 386 | DEBUG. By default, 386 | DEBUG is insensitive to the case of all program symbols. For example, the symbols "my\_sym", "my\_SYM," and "MY\_SYM" are all considered identical by default. The -TWOCASE switch enables case sensitivity in 386 | DEBUG. The -ONECASE switch disables case sensitivity. Since this is the default, this switch is redundant and is provided only for consistency.

The -TWOCASE switch should be used when debugging only if the -TWOCASE switch was used when the program was linked by 386 LINK.

#### Syntax:

- -ONECASE
- -TWOCASE

#### **Short Form:**

- -ONEC
- -TWOC

#### Example:

386debug -twocase hello

#### 2.2.3 Default Pointer Size Switches

The -SMALL, -COMPACT, -MEDIUM, and -LARGE switches specify the memory model of the program being debugged. This information is necessary for correct operation of the K and KA (stack trace) commands and the \* indirection operator.

The -SMALL switch specifies a small model program (NEAR code and NEAR data pointers). The -COMPACT switch specifies a compact model program (NEAR code and FAR data pointers). The -MEDIUM switch specifies a medium model program (FAR code and NEAR data pointers). The -LARGE switch specifies a large model program (FAR code and FAR data pointers). The memory model for a program is determined when the program is compiled.

The default setting, if no pointer size switch is given, is a small model program. When debugging protected mode programs, small model should always be selected, as 386 | DOS-Extender currently requires small model for protected mode programs.

### Syntax:

- -SMALL
- -COMPACT
- -MEDIUM
- -LARGE

#### **Short Form:**

- -SMALL
- -COMPACT
- -MEDIUM
- -LARGE

## Example:

386debug -medium hello 386debug -large hello

## 2.2.4 Loading a Different Symbol Table Switch

The -SYMFILE switch loads the symbol table out of an .EXP file other than the program file initially loaded. This feature is useful for debugging program overlays or add–ins loaded by the application program during execution. When the -SYMFILE switch is used the symbol table for the primary program (the program initially loaded) is not available.

If no file extension is given, 386 | DEBUG automatically looks for files with the extension .EXP, .EXE, or .REX.

Please see section 2.5 for additional information about loading a symbol table from an alternate .EXP file.

## Syntax:

-SYMFILE filename

#### **Short Form:**

-SF filename

#### **Example:**

386debug -sf display shell

## 2.2.5 386 DOS-Extender Switches

It is not possible to enter 386 | DOS-Extender switches on the command line. All switches entered before the program name are processed by 386 | DEBUG, not 386 | DOS-Extender.

386 | DOS-Extender switches specified at program link time are automatically recognized by 386 | DEBUG. Although most 386 | DOS-Extender switches that apply to a specific application rather than to a particular machine or environment can be selected at link time, some switches can only be used at run time.

To use 386 | DOS-Extender switches that can only be given at run time, follow one of these methods:

Place the switches in the environment variable, DOSX, which is always read by 386IDOS-Extender. After the debug session, remove the switches from the environment variable, so that the switches are not inadvertently left active when later running or debugging a different application. The following lines are an example of placing the switches in the environment variable:

```
set dosx=-weitek on
386debug -twocase hello
set dosx=
```

Configure the switches into 386IDEBUG itself, 386DEBUG.EXE, with the CFIG386 utility. By default, CFIG386 configures the switches into the stub loader on the front of 386DEBUG.EXE. When 386IDEBUG is running, the stub loader passes the configured switches to 386IDOS-Extender. After the debug session is complete, remove the configured switches or they will apply to all programs being debugged. Switches configured into 386IDEBUG apply only to programs being debugged, not to programs run directly with RUN386.EXE. The following lines are an example of configuring the switches into 386IDEBUG itself:

```
cfig386 386debug -debug 1
386debug -twocase hello
cfig386 -clear 386debug
```

Configure the switches directly into the 386IDOS-Extender file, RUN386.EXE, with the CFIG386 utility. Like the DOSX environment variable, switches configured directly into 386IDOS-Extender apply to all application programs, whether run with RUN386.EXE or debugged with 386DEBUG.EXE. The switches can be left in the file permanently, if desired. For example, if the 386IVMM virtual memory driver is always used when running or debugging applications, configure the -VMFILE switch into RUN386.EXE in the following manner:

```
cfig386 run386 -vmfile vmmdrv 386debug -twocase hello
```

## 2.3 Starting 386 | DEBUG

Programs to be debugged should be built with a symbol table containing all the public symbols in the program. This is done by using the -SYMBOLS switch on the command line when linking the program. The symbol table is appended to the end of the .EXP or .EXE file and is automatically loaded by 386 | DEBUG.

To begin debugging a program under the control of 386 | DEBUG, type a command line, as described in section 2.1. For example:

C>386debug hello

386 | DEBUG loads the program into memory and then displays a banner:

386|DEBUG 3.0--Copyright(C)1986-91 Phar Lap Software Inc. [2000 bytes in the symbol table]

[80386 protected mode]

The first line of the banner gives the version number of 386 | DEBUG that is running. The second line gives the size in bytes of the program's symbol table. The fourth line indicates if the application is running in real mode or in protected mode. A real mode application uses the 80386 CPU as if it were an 8086/8088 CPU. In protected mode, the full power of the 80386 CPU is available to an application.

After the banner is displayed, 386 | DEBUG prints a minus sign (-), which means it is ready for you to type the first debugger command. 386 | DEBUG prints a minus sign any time one command has been completed and it is ready for another.

## 2.4 Command Formats

386 | DEBUG commands consist of a command name followed by command operands. Multiple operands are separated by spaces or tabs. Throughout this manual, optional command operands are enclosed in brackets ([]). If an optional command operand is not specified, a default value for the unspecified operand is assumed. Default operands for each command are specified with the command description.

## 2.4.1 Command Operands

Operands to 386 | DEBUG commands are comprised of the following basic elements:

- The name of a public symbol whose value is to be used by the command.
- The name of an 80386 register. Depending on the context of the command in which it is used, a register name can either specify a register to be acted upon (e.g., by the "change register" command) or identify the register contents as the value to be used by the command (e.g., display memory pointed to by EBX in the data segment).
- An absolute number. All absolute numbers are interpreted as hexadecimal (base 16) by 386IDEBUG.
- A character string surrounded by single quotes (e.g., 'ABC').
- The colon (:) character. This character is used to enter addresses in segment:offset form (e.g., FS:100).
- The plus (+) and minus (-) characters. These characters are used to combine absolute numbers, symbols, and register values in simple expressions.
- The asterisk (\*) character. This character is used as an indirection operator in expressions.
- The left parenthesis ( ( ) and right parenthesis ( ) ) characters, used to parenthesize expressions.

#### **Example:**

-R EAX	Display contents of register EAX
-DB 100	Display memory at DS:100
-U main	Unassemble function "main"
-DD ptr_array	Display contents of "ptr_array" in doubleword
-DB *bufptr	format Display memory pointed to by variable "bufptr"
-DB FS:EBX	Display memory at FS: [EBX]
-F 0 50 'STK'	Fill memory with string, 'STK', from DS:0 to
-DB EBX+50	DS:50 Display memory at DS:[EBX+50]

## 2.4.2 Symbols

When a debug session is started, 386 | DEBUG reads the symbol table at the end of the .EXP or .EXE file into memory. This symbol table contains an entry for each public symbol in the program. In assembly language programs, symbols are declared public with the PUBLIC directive. In C

programs, all functions and global variables which have not been declared static are public symbols.

Each symbol is assigned a value which is equal to its address in memory. Whenever a symbol is used in a command, 386 | DEBUG substitutes the symbol value. Symbols can, therefore, be used wherever memory addresses are valid in debugger commands, and can also be combined in expressions. The X command lists all public symbols and their values, and can also be used to obtain the value of a specific symbol.

By default, 386 | DEBUG ignores case when looking up symbols. If the program has been linked with the -TWOCASE switch, use the 386 | DEBUG -TWOCASE switch to preserve upper and lower case for symbol lookup.

Symbol values in a real mode program are a full segment:offset address, where the offset is determined at program link time, and the segment paragraph address is determined when the program is loaded into memory. Symbol values in a protected mode program are only an offset in the single program segment constructed by 386 LINK.

Since all absolute numbers are entered in hexadecimal, 386 | DEBUG assumes that the letters A-F are used to form numbers. A symbol will therefore only be recognized by 386 | DEBUG if it contains at least one character that is not in the set {0-9,A-F}. For example, if a program has a public symbol named "ABC", it is not possible to reference that symbol because 386 | DEBUG interprets it as a hexadecimal number and does not attempt to look it up in the symbol table.

## Example:

```
-U chk_err Unassemble error check routine
-DB buf+40 Display memory 64 bytes into buffer "buf"
```

## 2.4.3 Expressions

Command operands may be specified as:

- a symbol name
- an absolute number

- a register whose contents are the value to be used
- an expression which combines symbols, absolute numbers, and register contents, and is resolved by 386IDEBUG to a single value.

All arithmetic in expressions is performed with values treated as 32-bit unsigned numbers.

There are four expression operators. The addition (+) and subtraction (-) operators are the only arithmetic operators. The indirection (\*) operator is used to perform one level of address indirection. The segment override (:) operator is used to separate the segment selector value from the offset in a full segment:offset address. The table below lists operator precedence, where operators of higher precedence are evaluated before operators of lower precedence. Operators of equal precedence are evaluated left to right. Please see the following table.

Operator	Precedence	
* (indirection)	highest	
: (segment override)	lowest	

Operator precedence may be overridden through the use of parentheses within the expression. Operations within parentheses are always evaluated before adjacent operations.

In the following example, the commands are sequential. Values shown on one line are used in the following lines.

#### **Example:**

-R EAX EAX=00000050	Display value of EAX
-R EBX EBX=00000100	Display value of EBX
-DB EBX	Display memory at DS:00000100
-DB EBX-EAX	Display memory at DS:000000B0
-DB CS:20+90-5	Display memory at CS:000000AB

-X bufptr bufptr = 00000400	List symbol "bufptr"
-DD bufptr bufptr+4 0014:00000400 00000800	Display pointer at "bufptr"
-DB *bufptr	Display memory pointed to by "bufptr" (at DS:00000800)
-DB *bufptr+100	Display memory at DS:00000900
-DD ES:0 4 004C:00000000 00000A00	
-DB CS:*(ES:0)	Display memory at CS:00000A00

#### 2.4.4 Addresses

Many 386 | DEBUG commands require the address of a memory location as a command operand. A memory address is specified with a segment selector value, and a byte offset within the segment. In protected mode, the segment selector is an index to an entry in the Local Descriptor Table (LDT) or the Global Descriptor Table (GDT) which describes the segment. In real mode, the segment selector is the paragraph base address of the segment in memory.

A 386 | DEBUG address operand can be specified as either a simple offset or a full segment:offset address. An offset is specified as a hexadecimal number, or by using a program symbol when debugging a protected mode program. If the expression specifies only an offset, a default segment is chosen based on the command. The U (unassemble), G (go), BP (set breakpoint), T (trace), TQ (trace quietly), P (procedure trace), and PQ (Ptrace quietly) commands all assume the segment pointed to by the CS register as the default segment. All other commands assume the segment pointed to by the DS register as the default.

A full segment:offset address can be specified in an expression in one of several ways:

- When debugging a real mode program, using a program symbol specifies the segment in which the symbol is defined.
- If the contents of an 80386 register is used in an expression, the default segment is the same as that assumed by the 80386 processor. Specifically, addresses formed with the contents of ESP, EBP, SP, or BP assume the SS segment, addresses formed with EIP or IP assume the CS segment, and all other registers assume the DS segment.
- A segment selector value can be explicitly specified using the : segment override operator.
- When the \* indirection operator is used in an expression, the resulting address value includes a segment selector if a FAR pointer is retrieved from memory. If a NEAR pointer is retrieved, the address value is only an offset.

The expression operators and the ways in which they can be used to create memory addresses are described in more detail in section 2.4.6.

## Example:

-DB 100	Display memory at DS:100
-G = 100	Go (start executing) at location CS:100
-DB EBX	Display memory at DS:[EBX]
-DB ESP	Display memory at SS:[ESP] (top of stack)
-U *ESP	Unassemble at procedure return address
	on top of stack
-G =fatal err	Go (start executing) at routine "fatal_err"
-DB buf	Display contents of buffer "buf"
-DB buf+EBX	Display memory EBX bytes into "buf"
-DB FS:EBP+20	Display memory at FS: [EBP+20]
-DB 3C:0	Display memory at 003C:00000000

## 2.4.5 Ranges

Several commands operate on memory locations within a range of addresses. A range is defined by its start and end addresses. Most commands that require a range permit you to specify only the start address of the range, and assume a default end address (typically, a specific number of bytes past the start address). In many cases, the start address of the range is also optional, and the command will supply a default value one byte past the end of the range specified the last time the command was used, or zero if the command has never been used in the debugging session.

The command descriptions use the following notation with commands that take an address range as operands:

range If both the start and end addresses must be specified; address | range If the start address is required, but the end address is optional;

[range] If both the start and end addresses are optional.

When specifying an address range, a full segment:offset address may only be specified for the start address. The end address for a range is required to be in the same segment as the start address, and a segment override for the end address is therefore prohibited.

## Example:

-DB buf buf+100
Display the first 256 bytes of "buf"
DB 100 200
Display memory from DS:100 to DS:200
Display memory from CS:10 to CS:20
Display memory from DS:0 to DS:7F

(128 bytes)
Display memory from DS:80 (one past the end of the last DB command) to DS:FF

## 2.4.6 Expression Operators

## Addition and Subtraction Operators

The + and - operators are used to perform arithmetic addition or subtraction of two operands. When a memory address is being formed, at most one operand can be a full segment:offset address; the other operand must be a simple offset (an absolute number). The only exception is when a symbol is combined with the contents of a register. In that case, the specified segment selectors may differ, and the segment in which the symbol is defined takes precedence over the default segment assumed for the register.

## **Example:**

-DB 50+70-10 Display memory at DS:B0
-T 3+5 Trace 8 instructions

-DB EBP+10 Display memory at SS:10[EBP]

-DB buf+BP Display memory BP bytes into "buf"

## Segment Override Operator

The : segment override operator is used to override the segment selector part of a memory address. The segment selector value to the left of the : operator replaces the segment selector specified by the expression to the right of the : operator. This operator is useful when the default segment selected by 386 | DEBUG for a command is not the desired segment.

#### Example:

-DB CS:20 Display memory in the code segment

-DB FS:EBP Display memory at FS:[EBP]
-G =4C:10 Start execution at 004C:10

## **Indirection Operator**

The indirection operator (\*) is used to perform one level of address indirection through a pointer stored in memory. Using this operator, a pointer stored in memory can be used to directly access the memory locations to which it points, instead of having to enter two commands, one dumping out the pointer, and the second using the hexadecimal value of the pointer obtained from the first command.

The \* operator uses the memory model of the program to decide whether to pick up a NEAR (offset only) or a FAR (segment:offset) pointer out of memory. The program's memory model is specified with the -SMALL, -COMPACT, -MEDIUM, or -LARGE command line switches, or with the MS, MC, MM, or ML debugger commands.

For memory models (such as COMPACT or MEDIUM) which use one type of pointer for data and the other type for code, the \* operator decides if it is picking up a code or a data pointer based on whether the default segment

for the command is the code or data segment. For example, if the command is U (unassemble), the \* operator will pick up a code pointer; if the command is DB (display memory), the \* operator will pick up a data pointer. The default segment for each command is specified in section 2.4.4.

The operand of the \* operator is the memory address at which the pointer is stored. As usual, the address of the pointer can be specified as either a full segment:offset, or as an offset only. If only an offset is specified, the default segment for 386 | DEBUG command is used to locate the segment in which the pointer is stored.

The pointer address retrieved from memory completely replaces the address used to retrieve the pointer. If a NEAR pointer is retrieved, the resulting address specifies an offset value only, regardless of whether the memory address at which the pointer is stored specifies a segment selector.

## Example:

-DB *bufptr	Display memory pointed to by variable "bufptr"
-S *bufp *bufp+400 FF	Search array pointed to by "bufp" for bytes with values FF
-U *(EBP+4)	Unassemble instructions at return address from the current routine
-DD FS:10 14 0054:00000010 000002A0	Display NEAR pointer value
-DB *(FS:10)	Display memory at DS:2A0
-DB FS:*(FS:10)	Display memory at FS:2A0

## 2.5 Loading Alternate Symbol Tables

The -SYMFILE switch loads the symbol table out of an .EXP file other than the program file initially loaded. The syntax of the switch is:

-SYMFILE filename

The *filename* parameter specifies the executable file from which to load the symbol table. Use this switch for debugging program overlays or add-in drivers loaded by the main application program during execution. When this switch is used, the symbol table for the primary program (the program initially loaded) is not available.

Programs that load in add-in drivers or overlays at runtime often allocate separate code and data segments (i.e., they don't load the add-in into segments 000Ch and 0014h). Since the selectors for these additional code segments and data segments are not known until runtime, you must tell 386 | DEBUG which segment selectors are used for the symbol table loaded with -SYMFILE.

Use the XR command (see section 3.30) to tell 386 | DEBUG which segments are in the symbol table. The XR command must be given after the add-in driver or overlay is loaded.

For example, suppose your main program, SHELL.EXP, loads a secondary program called DISPLAY.EXP. To debug DISPLAY.EXP symbolically, follow these steps:

Use the -SYMFILE switch to load the symbol table for DISPLAY.EXP rather than SHELL.EXP:

386debug -symfile display shell

Set a breakpoint in shell (the main program) following the routine that loads DISPLAY.EXP. Since the symbol table for shell is not loaded, you have to use hex addresses obtained from the map file for shell. (Alternatively, you could assemble an INT 3 instruction into your shell program to pass control to 386IDEBUG). Let's assume the load routine for shell is at hex address 8F90. Then you can get control after DISPLAY.EXP is loaded with these commands:

-g 8F90

-g \*esp

Use the XR command to tell 386IDEBUG which segments are used for DISPLAY.EXP, so it uses those segments (rather than 000Ch and 0014h) for symbols. SHELL.EXP could print out the segment selectors, or you could use the DL command to examine the LDT before and after DISPLAY.EXP is loaded to see which segment selectors are allocated. Let's assume the code segment for display is 007Ch, and the data segment is 0084h. Use the XR command to relocate the symbol table to these selectors:

-xr C 7C

## 2.6 Controlling 386 | DEBUG Display Output

Some 386 | DEBUG commands can potentially generate a very large amount of output to the display (for example, displaying a huge block of memory). When this occurs, it is possible to stop (temporarily) output and prevent desired information from scrolling off the top, and to cancel output to the display and return to 386 | DEBUG command prompt.

The CTRL/S character (hold down the CTRL key while striking the S key) temporarily stops output to the display. After examining the information on the display, any key can be pressed to cause 386 | DEBUG to resume output to the display.

The CTRL/C character cancels the output for a command and returns to the 386 | DEBUG command prompt.

## 2.7 Input and Output Redirection

## 2.7.1 I/O Redirection to a Communications Port

386 | DEBUG supports redirection of keyboard and display I/O to a terminal connected to communications port 1 or 2 of the PC. This feature is useful for debugging I/O-intensive applications that make heavy use of the system display and/or the keyboard.

I/O can be redirected to one of the communications ports either by command line switches when the debugging session is started, or by debugger commands entered during the debug session. If the -COM1 or

-COM2 command line switch is given, then I/O, including 386 | DEBUG startup banner, is redirected to the specified communications port. If the -BAUD switch is also used, then the port will be reprogrammed to use the specified baud rate, 8 data bits, one stop bit, and no parity. If the -BAUD switch is not given, then the communications port parameters are not modified. This allows arbitrary parameters to be specified with the DOS MODE command before starting the debug session.

At any point during a debug session, I/O can be redirected to a communications port with the COM1 or COM2 commands. These commands do not modify the port parameters; they just redirect 386 | DEBUG I/O. When entering commands at a remote terminal, the CON command can be used to redirect I/O back to the system console.

When I/O is redirected to a communications port, CTRL/C usage is slightly modified. While the application program is running and making DOS calls, a CTRL/C typed at the remote terminal will be ignored, and a CTRL/C typed at the system keyboard will cause 386 | DEBUG to gain control (with I/O still redirected to the communications port). When 386 | DEBUG has control, a CTRL/C typed at the remote terminal causes output (such as a long dump) to be terminated. When 386 | DEBUG has control and is waiting for a command, a CTRL/C typed at the system keyboard redirects I/O to the system console. This last feature is useful if you are unable to enter commands at a remote terminal because of an incorrect baud rate or similar problem. Please see section 2.8.5 for more information on CTRL/C processing.

When I/O is redirected to a communications port, there is no type-ahead feature. Characters typed when 386 | DEBUG is not waiting for input will be ignored.

## 2.7.2 I/O Redirection to Disk Files

Normally, 386 | DEBUG commands are typed at the keyboard, and output is printed on the display. However, it is possible both to read commands from a script file on the disk, and to redirect output to a log file on the disk.

Command input redirection is specified on the command line by typing the less than character (<), followed by the name of the file containing the commands. Blank lines in the input file are ignored, and the special character \* may be used at the beginning of a line to denote a comment (a line to be ignored by 386 | DEBUG). Commands are processed up to the first quit (Q) command in the file, or to the end of the file, whichever comes first. If there is no quit command in the file, 386 | DEBUG quits when the end of the file is reached.

Output redirection is specified on the command line by typing the greater than character (>), followed by the name of the file to which the output is written.

Note that although it is possible to redirect output without redirecting input, the results can be confusing if you do so. When output is redirected, all characters which would be displayed on the screen are instead written to the output file. This includes keystrokes which are normally echoed on the screen. Thus, it is impossible to see what is being typed on the keyboard if input is not redirected while output is redirected.

#### Example:

To type commands at the keyboard and to print debugger output on the display:

386debug hello

To read commands from the file "script" and to print debugger output on the display:

386debug hello <script

To read commands from the file "script" and to write debugger output to the file "log":

386debug hello <script >log

## 2.8 Regaining Control

386 | DEBUG commands may be entered only when 386 | DEBUG has control and has issued the dash (-) as a prompt. There are a variety of ways to regain control in 386 | DEBUG when the application program is executing. These methods are discussed in the sections below. Whenever 386 | DEBUG regains control, it will print out the reason it regained control and the current state of the processor.

## 2.8.1 Program Breakpoints and Instruction Tracing

Program breakpoints cause 386 | DEBUG to stop execution of the application immediately before it executes the instruction at a specified breakpoint address. After gaining control, 386 | DEBUG displays the 80386 registers and prompts for a command. Breakpoints allow a program to execute up to the point where an error is suspected, in order to examine memory and registers for the cause of the error.

There are two kinds of breakpoints: temporary breakpoints and permanent breakpoints. Temporary breakpoints are set with the go (G) command and are always cleared when 386 | DEBUG gains control again for any reason. For a temporary breakpoint to remain in effect, it must be specified each time the G command is entered. Permanent breakpoints are set with the BP command, and remain in effect until cleared with the BC command or disabled with the BD command. Permanent breakpoints are useful, for example, to set a breakpoint on a critical function that may be called many times. Temporary breakpoints are useful when stepping through a program, since there is no need to clear temporary breakpoints.

Breakpoints must be set on instruction boundaries. If breakpoints are placed in the middle of an instruction, they will not be recognized and 386 | DEBUG will not gain control. The unassemble (U) command can be used to determine the address of a specific instruction.

A feature related to breakpoints is the trace (T) command. This command executes a single instruction in the program under test and then returns control to 386 | DEBUG. It has the same effect as using the G command to set a breakpoint on the next instruction. This feature is useful for stepping through a sequence of instructions in which a bug is suspected.

#### Example:

Set a permanent breakpoint at print routine
Set a temporary breakpoint at fatal error
routine and execute from current address
Set temporary breakpoints at CS:55 and CS:7B
and execute from current address
Trace five instructions

## 2.8.2 Data Watchpoints

Data watchpoints cause 386 | DEBUG to stop execution of the application immediately after execution of an instruction that reads or writes a specific memory location. When a data watchpoint occurs, 386 | DEBUG displays the memory location that was accessed and the current register contents, and then prompts for a command. Data watchpoints can be useful for finding bugs which cause code or data locations in the program to be corrupted.

Watchpoints are set with the WP command. When a watchpoint is set, both the address of the location to be watched and the size of the expected access (one, two, or four bytes) must be specified. If the wrong data size is used, the hardware will not detect an access to the memory location. In addition, watchpoints can be set to occur on a write access only, or for either a read or a write access to the memory location. By default, WP sets a write only watchpoint. To set a read/write watchpoint, the character r must be the last parameter given to the WP command.

## Example:

-WP bufp 4	Set watchpoint to occur when pointer variable
	"bufp" is modified
-WP errf 2 r	Set watchpoint to occur when an error flag is
	either read or written
-WP main+20 1	Set watchpoint to occur when code in the
	"main" routine gets wiped out

## 2.8.3 Processor Exceptions

The 80386 processor can detect certain classes of program errors. When one of these errors occurs, the CPU will generate a processor exception.

Interrupt numbers 00 through 20 (hex) are reserved by Intel for processor exceptions. 386 | DEBUG installs handlers for most of the 80386 processor exceptions and will therefore gain control whenever an exception occurs. The 80386 processor exceptions are:

Number (hex)	<u>Error</u>
00	Divide overflow
01	Debugger trace and breakpoints
02	External non-maskable interrupt
03	INT 3 debugger breakpoint
04	Overflow (INTO instruction)
05	BOUND instruction (array bounds checking)
06	Illegal instruction opcode (typically caused by a program jumping into data)
07	Numeric coprocessor not available
08	Double fault
09	Coprocessor segment overrun (coprocessor attempts to access data past the end of a segment)
0A	Invalid TSS (task state segment used in task switch invalid)
0B	Segment not present (attempt to use segment descriptor marked not present in GDT or LDT)
0C	Stack fault (segment limit violation on operations that use the SS segment register)
0D	General protection violation
Œ	Page fault (page not present)

## 2.8.4 Program Termination

When the program being tested terminates, 386 | DEBUG gains control. The register contents are not displayed, as 386 | DEBUG does not know the location from which the DOS exit call was made or the register values at the time of the call. (386 | DEBUG does not gain control until after the DOS program termination code has executed.) Instead, the exit value passed to DOS by the program is displayed.

After the program terminates, any execution command such as go (G), trace (T), or procedure trace (P) should not be executed. This restriction

exists because the memory allocated to the program has already been freed, and thus there is no valid instruction or stack space. Using one of these switches after program termination will generate the message "Application program not in memory".

#### 2.8.5 CTRL/C

Programs that make DOS calls may be interrupted by typing CTRL/C. When DOS gains control and sees a CTRL/C in the keyboard buffer, it issues interrupt number 23 (hex), which is also handled by 386 | DEBUG. This is useful for interrupting programs caught in an infinite loop.

There are some differences in the behavior of CTRL/C between debugging real mode programs and protected mode programs. If the program is executing in protected mode when it issues the DOS call which results in a CTRL/C interrupt, then the register display will show the register contents at the time the DOS call is issued. After gaining control via a CTRL/C from a protected mode program, 386 | DEBUG will not permit any register contents to be altered, because when control is passed back to the program, the DOS call actually completes. Commands that do not alter register values, however, can be used without restriction.

When a real mode program is interrupted with CTRL/C, the register display shows the register contents at the time DOS issues the CTRL/C interrupt. 386 | DEBUG permits registers to be modified, though this is not advised because it affects how DOS finishes processing the system call which is interrupted. The trace (T) command causes 386 | DEBUG to trace instructions in DOS rather than in the application program (unlike protected mode programs).

## 2.9 Protected Mode versus Real Mode Debugging

386 | DEBUG can be used to debug both real mode and protected mode programs. Real mode programs have a default file extension of .EXE, and protected mode programs have a default file extension of .EXP. The banner displayed by 386 | DEBUG tells whether the program runs in protected mode or real mode.

386 | DEBUG provides the same basic commands for debugging in either mode. The only commands that are not available when debugging real mode programs are the commands to display protected mode system tables: DG, DL, and DI. All other commands operate identically in either mode.

The main difference between debugging real mode and protected mode programs is in the values used for segment selectors. For protected mode programs, segment selectors are an index into the LDT (local descriptor table), which can be displayed with the DL command, and therefore tend to be small numbers, such as hexadecimal "0C" for the code segment and hexadecimal "14" for the data segment. For real mode programs, segment selector values are the paragraph base address of the segment and thus tend to be large numbers, such as hexadecimal "3D69". For a more detailed description of the differences between real mode and protected mode programs, please see the 386 | ASM Reference Manual.

For protected mode programs, symbol values are 32-bit offsets in the single program segment that is pointed to by both CS and DS. For real mode programs, symbol values are in segment:offset form, where the segment value is the 16-bit paragraph base address in memory of the segment in which the symbol is located, and the offset value is a 16-bit offset within the segment.

## Examples of protected mode commands:

DB	buf	Display	data	in	buffer	"bu:	E"			
DB	CS:100	Display	data	at	offset	100	in	segment	pointed	to
		by CS								
DB	C:100	Display	data	at	offset	100	in	segment	whose	
		selector	r vali	ie :	is "OC"					

#### Examples of real mode commands:

					segment	pointed	to
DB	3D69:100				segment		



# 386 | DEBUG Command Reference

This chapter provides a detailed reference guide to 386 | DEBUG commands. For each command, the following information is given:

- The syntax of the command. Optional operands are in brackets ([]);
- A description of the command;
- Some example usages of the command.

## 3.1 BC, BD, BE, BL, and BP Commands — Breakpoint Control

## Syntax:

BC breakpts

BC \*

BD breakpts

BD \*

BE breakpts

BE \*

BL

BP address

## Description:

The breakpoint control commands set, clear, enable, disable, and list breakpoints. Breakpoints must be set on instruction boundaries. They cause 386 | DEBUG to gain control before the instruction is executed.

The BC command clears breakpoints. It takes as an argument either a list of breakpoint numbers, or the special character \*, which is a shorthand for all breakpoints. Once a breakpoint has been cleared, it is no longer listed by the BL command and cannot be enabled with the BE command. If the

breakpoint is needed again after being cleared, it must be reset with the BP command.

The BD and BE commands disable and enable breakpoints. They take as an argument either a list of breakpoint numbers, or the special character \*, which is a shorthand for all breakpoints. The BD command is used to temporarily disable breakpoints that are not desired to be active. The BE command is used to enable breakpoints previously disabled with the BD command.

The BL command lists breakpoints set with the BP command. Each breakpoint is identified by a number assigned by 386 | DEBUG. For each breakpoint, the BL command lists its breakpoint number, the address of the instruction on which the breakpoint is set, and the internal mechanism that 386 | DEBUG used to set the breakpoint (either "Debug reg" if one of the 80386 hardware registers was used, or "INT 3", if the 1-byte debug interrupt instruction was used). In addition, if the breakpoint has been disabled, the BL command prints "\*\*\*disabled\*\*\*" following the breakpoint information.

The BP command sets breakpoints. The command operand is the address of the instruction on which to set the breakpoint. Breakpoints set with the BP command may be either enabled (on) or disabled (off). When the breakpoint is first set with the BP command, it is always enabled.

## Example:

```
-BP main
-BP err_chk
-BP main+9
-BL
0 MAIN mechanism=Debug reg
1 ERR_CHK mechanism=Debug reg
2 MAIN+09 mechanism=Debug reg
-BD 0 2
-BL
0 MAIN mechanism=Debug reg ***disabled***
1 ERR_CHK mechanism=Debug reg
2 MAIN+09 mechanism=Debug reg ***disabled***
```

-BE \*

-BC 1

-BL

0 MAIN mechanism=Debug reg

2 MAIN+09 mechanism=Debug reg

## 3.2 C Command — Compare Memory

#### Syntax:

C range address

## Description:

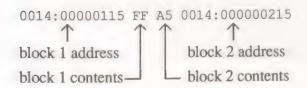
The compare memory command compares two blocks of memory to determine if they are equal. Any differences between the two blocks are printed on the screen. The C command has three operands. The first two operands specify the address range of the first block to be compared. The last operand is the start address of the second block of memory. No end address for the second block may be specified, since 386 | DEBUG always assumes the second block is the same size as the first block. If no segment selector is specified for either the first or second block, the value in the DS register is used by default.

## Example:

The command

-C 100 1FF 200

compares memory locations 100 through 1FF, inclusive, on a byte-by-byte basis with memory locations 200 through 2FF. For each pair of bytes that differ, 386 | DEBUG displays a line containing the address and value of each byte. It is formatted as follows:



The command

-C bufl bufl+100 buf2

compares the first 256 bytes of buffers "buf1" and "buf2".

## 3.3 COM1, COM2, and CON Commands — I/O Redirection

#### Syntax:

COM1

COM2

CON

## Description:

The I/O redirection commands redirect 386 | DEBUG I/O between the system console and a terminal connected to communications port 1 or 2 on the PC. The COM1 command is entered at the system keyboard to redirect I/O to communications port 1. The COM2 command is entered at the system keyboard to redirect I/O to communications port 2. The CON command is entered at the remote terminal connected to a communications port to redirect I/O back to the system console.

Note that the communications port parameters are not modified by the COM1 or COM2 commands. The port parameters can be set explicitly by the DOS MODE command before starting the debugging session. Alternatively, the -BAUD command line switch can be used in conjunction with the -COM1 or -COM2 command line switch to reprogram the communications port to the specified baud rate, 8 data bits, one stop bit, and no parity. I/O can then be redirected back and forth with the CON and COM1 or COM2 commands without changing the port parameters.

If you are unable to enter commands at a remote terminal after redirecting I/O because of incompatible baud rates or some other problem, type a CTRL/C at the system keyboard to redirect I/O back to the system console. This has the same effect as entering the CON command from the remote terminal.

## **Example:**

To redirect I/O to communications port 1 at the main system:

-COM1

To redirect I/O from a remote terminal back to the console at a remote system:

-CON

## 3.4 D, DA, DB, DD, and DW Commands — Display Memory

#### Syntax:

D [range]

DA [range]

DB [range]

DD [range]

DW [range]

## Description:

The display memory commands display memory locations on the screen. They take an optional range operand, which specifies the block of memory to be displayed. If no start address is specified, memory is displayed at the location following the last byte displayed with the most recent display command. If no end address is specified, 128 bytes of memory are displayed.

The DA command displays data in ASCII format only. It displays data up to the first zero byte encountered, or up to the endpoint of the specified range if no zero byte is encountered.

The DB command formats the display as individual bytes. The data bytes are displayed in both hexadecimal and ASCII format. Non-printable ASCII characters are displayed as a period (.).

The DD command formats the display as hexadecimal doublewords (32-bit values).

The DW command formats the display as hexadecimal words (16-bit values). This is useful because the DB command shows the bytes of word values in reverse order, due to the byte ordering imposed on values by the 80386.

The D command is simply a shorthand for the most recently used display command. For example, if a DD command is given followed by a D command, the display is formatted as doublewords.

If the previous display command was a floating point display (DF, DS, DQ or DT) the D command repeats the floating point display format.

If no segment is specified with the address, the display memory commands default to the segment pointed to by DS.

## Example:

```
-DB buf buf+F
0014:00000100 45 72 72 6F 72 00 01 02-12 64 00 00 00 00 00 00 Error...d.....
-DW buf buf+F
0014:00000100 7245 6F72 0072 0201 6412 0000 0000 0000
```

```
-DD buf buf+F 0014:00000100 6F727245 02010072 00006412 00000000
```

```
-DA buf buf+F
0014:00000100 Error
```

-DW CS:0 F 000C:00000000 139A 0000 0005 0200 A5A5 0090 001A 0000

## 3.5 DF, DS, DQ, and DT Commands — Display Memory as Floating Point

#### Syntax:

DF [range]
DS [range]
DQ [range]
DT [range]

## Description:

The DF, DS, DQ, and DT commands display memory as IEEE (the floating point format used by the 80387, the Weitek 3167/4167, and the Cyrix EMC87 coprocessors) floating point numbers. They take an optional range operand, which specifies the block of memory to be displayed. If no start address is specified, memory is displayed at the location following the last byte displayed with the most recent display command. If no end address is specified, a single floating point number is displayed.

The DF and DS commands display 4-byte (short) floating point numbers. These are values declared with the "float" data type in C programs, or with the DD directive in assembly language code.

The DQ command displays 8-byte (quadword) floating point numbers. These are values declared with the "double" data type in C programs, or with the DQ directive in assembly language code.

The DT command displays ten-byte floating point numbers. These are values declared with the DT directive in assembly language code.

Please see the D command, section 3.5, to repeat any display format.

#### **Example:**

```
-ds flt_const flt_const+c
0014:00002440 B4 C8 16 40 2.3559999E+00
0014:00002444 D0 0F 49 40 3.1415901E+00
0014:00002448 AC 8B 5B 3C 1.3400000E-02
-df
0014:0000244C AE C7 10 44 5.7912000E+02
```

## 3.6 DG, DI, and DL Commands — Display System Tables

## Syntax:

DG [range]
DI [range]
DL [range]

## Description:

The DG, DI, and DL commands display the system GDT (global descriptor table), IDT (interrupt descriptor table), and LDT (local descriptor table), respectively. Each command takes two optional operands: DI takes starting and ending interrupt numbers, and DG and DL take starting and ending segment selector numbers.

If no operand is given, 16 table entries are displayed, beginning with one past the last entry displayed the last time the command was used. If only a start operand is given, 16 entries are displayed, beginning with the one at the specified operand. If both a start and an end operand are given, all entries between the start and the end operand, inclusive, are displayed.

The GDT maps segments used by 386 | DOS-Extender, and the LDT maps segments used by the application program and 386 | DEBUG. Specific segment assignments are described in detail in the 386 | DOS-Extender Reference Manual. For each entry in the GDT or LDT, the DG or DL command displays the following information. All values are given in hexadecimal.

- The segment selector used to access the segment;
- The segment linear base address;
- The segment limit (in bytes);
- The flags byte (access rights byte) in the segment descriptor;
- The segment use type USE16 for a 16-bit segment, or USE32 for a 32-bit segment.

For a complete description of segment descriptors, including the bit definitions of the flags byte, please see the Intel manual, 80386 Programmer's Reference Manual.

The IDT contains descriptors which point to the interrupt service routines for each of the 256 possible interrupts supported by the 80386. For each entry in the IDT, the DI command displays the following information. All values are given in hexadecimal.

- The interrupt number that corresponds to the IDT entry (this is the same as the index of the entry in the table);
- The gate type;
- The segment selector and offset of the handler routine for the interrupt;
- The flags byte in the descriptor;
- The type;
- The descriptor privilege level.

The DG, DI, and DL commands are not available when debugging real mode programs.

## **Example:**

```
-DG 0 8
#0000 Segment not present
#0008 Base=00025900 Limit=0000FFFF Flags=9B USE32

-DL 4 C
#0004 Base=00044D60 Limit=000000FF Flags=92 USE32
#000C Base=00400000 Limit=00345FFF Flags=9A USE32
```

```
Flags=8E TYPE=386 DPL=00
#0000 INTGATE
              0064:00001FF0
                             Flags=8E TYPE=386 DPL=00
#0001 INTGATE
              0064:0000201F
                             Flags=8E TYPE=386 DPL=00
#0002 INTGATE 0064:00005A79
#0003 INTGATE 0064:0000208D
                             Flags=8E TYPE=386 DPL=00
                             Flags=8E TYPE=386 DPL=00
#0004 INTGATE 0064:000020C6
#0005 INTGATE 0064:000020F5
                             Flags=8E TYPE=386 DPL=00
#0006 INTGATE 0064:00002124
                             Flags=8E TYPE=386 DPL=00
                             Flags=8E TYPE=386 DPL=00
#0007 INTGATE 0064:00005C08
                             Flags=8E TYPE=386 DPL=00
#0008 INTGATE 0064:00002182
                             Flags=8E TYPE=386 DPL=00
#0009 INTGATE 0064:000021B1
#000A INTGATE 0064:000021E0
                             Flags=8E TYPE=386 DPL=00
#000B INTGATE 0064:0000220F
                             Flags=8E TYPE=386 DPL=00
                             Flags=8E TYPE=386 DPL=00
#000C INTGATE 0064:0000223E
#000D INTGATE 0064:0000226D
                             Flags=8E TYPE=386 DPL=00
#000E INTGATE 0064:0000229C
                             Flags=8E TYPE=386 DPL=00
#000F INTGATE 0064:00005A79
                             Flags=8E TYPE=386 DPL=00
```

## 3.7 DTSS, DTSS16, and DTSS32 Commands — Display Task State Segment

## Syntax:

DTSS [selector]
DTSS16 address
DTSS32 address

## Description:

The DTSS commands display task state segments. Most DOS application programs do not use task state segments, but embedded applications that are prototyped under DOS may use TSSs. See the Intel manual, 80386 Programmer's Reference Manual, for a description of the contents and use of a task state segment.

The DTSS command displays the contents of the task state segment specified by the segment selector. The descriptor corresponding to the selector must be a 386 or a 286 TSS system segment. If no selector is given, the selector in the task register (TR) is assumed.

The DTSS16 and DTSS32 commands assume the address specified with the command is the address of a TSS data structure. The DTSS16 command displays a 286 TSS, and the DTSS32 displays a 386 TSS.

## Example:

```
-DTSS 80
TSS address = 0080:00000000
EAX=00000000 EBX=00000000 ECX=00000000
                                      EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00000000
DS=0000 SS=0000 ES=0000 FS=0000 GS=0000
CS:EIP=0000:00000000 EFLAGS=00000000
SS:ESP(2)=0000:00000000 LDTR=0028 CR3=003D5000
                   IOMAP=0000
BACKLNK=0000
            TBIT=0
-DTSS32 88:0
TSS address = 0088:00000000
EAX=00000000 EBX=00000000 ECX=00000000
                                      EDX=00000000
                        EBP=00000000 ESP=00000000
ESI=00000000 EDI=00000000
DS=0000 SS=0000 ES=0000 FS=0000 GS=0000
CS:EIP=0000:00000000 EFLAGS=00000000
SS:ESP(0)=0120:00000100 SS:ESP(1)=0000:00000000
SS:ESP(2)=0000:00000000 LDTR=0028 CR3=003D5000
BACKLNK=0000 TBIT=0 IOMAP=0000
```

## 3.8 E, EB, ED, and EW Commands — Enter (Change) Memory

## Syntax:

```
E address value(s)
EB address value(s)
ED address value(s)
EW address value(s)
```

## Description:

The enter commands change memory locations. The E and EB commands are synonymous and are used to change memory one byte at a time. The EW command changes memory one word at a time, and the ED command operates on doublewords.

The operands to the enter commands are the address of the memory to modify, followed by a list of one or more values to enter at that memory location. If no segment selector is given with the address, the segment pointed to by DS is assumed.

The E and EB commands accept character strings as well as hexadecimal values to enter into memory. Each character in the strings is stored in memory as a separate byte.

## Example:

## 3.9 F Command — Fill Memory

## Syntax:

F range value(s)

## Description:

The fill command initializes a block of memory to a particular value. The command operands to the F command are a memory range, followed by one or more data bytes. The range specifies which memory locations are to be filled. The values must be either byte values or character strings, and are replicated as many times as necessary to fill the specified range. If no segment is specified with the address, the segment pointed to by DS is assumed.

## **Example:**

The command

-F 100 1FF 0

sets all memory locations from location DS:100 to 1FF, inclusive, to zero.

The command

-F 1000 1200 'HELLO'

fills locations DS:1000 to 1200, inclusive, with the string 'HELLO' repeated over and over.

## 3.10 G Command — Go

#### Syntax:

G [=address] [address(es)]

## Description:

The go command starts a program running. In its simplest form, the G command takes no operands. It starts execution at the instruction pointed to by CS:EIP (the instruction pointer). Control is returned to 386 | DEBUG when one of the conditions described in section 2.8 occurs.

The G command can also transfer control to an arbitrary location in the program. The start address to which control is to be transferred is indicated by preceding the address with an equal sign (=).

The G command also allows temporary breakpoints to be set in a program. Temporary breakpoints are specified by listing, as command operands, the address of one or more instructions where the breakpoints are to be set. Up to four temporary breakpoints can be set at any one time, but the program will run only until the first breakpoint is encountered. Any other breakpoints are cleared when 386 | DEBUG regains control. If you want a temporary breakpoint to remain set, you must re-enter it with the next G command.

If no segment selector is specified with either the breakpoint addresses or the start address, then the selector in the CS register is used by default.

## Example:

To continue execution at the current CS:EIP:

-G

To set a temporary breakpoint at function "rd\_line":

-G rd\_line

To continue execution at 000C:20:

-G = C:20

To continue execution at "fatal\_err" and set temporary breakpoints at "printf" and "exit":

-G =fatal\_err printf exit

## 3.11 H Command — Perform Hex Arithmetic

## Syntax:

H value value

## Description:

The hex arithmetic command provides a hexadecimal calculator in 386 | DEBUG. The command takes two 32-bit hex numbers as operands. It computes the sum of and difference between the two numbers, and the product and quotient, and displays the result in hex. The product and quotient will not be displayed if one of the numbers is zero.

## Example:

## 3.12 I Command — Input from Port

#### Syntax:

I port

## Description:

The input command is used to read a byte value from an input port. The single operand to the command is the number of the input port to read. 386 | DEBUG reads the port and displays the value read in hexadecimal.

#### Example:

-I 0Port 0000 = 00

## 3.13 K and KA Commands — Stack Trace

## Syntax:

K [arg\_count]

KA

## Description:

The stack trace commands display the current procedure call stack. The first line of the display shows the current procedure and offset within the procedure. Succeeding lines of the display trace the call back to the top level procedure, one line per procedure call. For example, the second line shows the procedure which called the current procedure, and the address of the instruction that will be executed when the current procedure returns.

Note that the stack trace commands rely on a standard stack frame being set up for each procedure. Many compilers optimize procedures which have no local variables by not setting up a stack frame. If this occurs, the stack trace commands will not display complete information — one or more of the links in the trace will be missing. To guarantee that the stack

trace command gives complete information, compiler switches should be used to force a standard stack frame to be set up for every procedure.

For the MetaWare High C or Professional Pascal compilers, put the statement "pragma off(optimize\_fp);" in a profile file or at the top of each source file.

For the Microway NDP C-386 and NDP Fortran-386 compilers, use the "-ga" command line switch when you compile the program.

For the Watcom C-386 compiler, use the switch "/3S" when you compile the program.

The K stack trace command does not attempt to determine how many arguments are passed to each procedure. An optional number can be given as a command operand, specifying the number of arguments to list for each procedure. If no operand is given, the K command displays no procedure arguments. Note that procedure arguments are displayed as 32-bit values when debugging protected mode programs, and as 16-bit values when debugging real mode programs.

The KA stack trace command with arguments attempts to automatically determine how many arguments each procedure has, so that it can display the correct number of arguments to each procedure. Since this command decides how many arguments to display automatically, it takes no command operands. Note that the KA command relies on the calling procedure popping arguments off the stack immediately after the procedure call. This command should not be used with languages (such as Pascal) which use the convention that the called procedure pops its arguments before returning. In addition, any compiler optimizations that delay popping of arguments should be disabled when using the KA command. For the MetaWare High C compiler, use the statement "pragma off(postpone\_callee\_pops);" in a profile file or at the top of each source code file.

In order to work correctly, the K and KA commands must know the memory model of the program being debugged. The memory model is specified with the -SMALL, -COMPACT, -MEDIUM, or -LARGE command line switches, or with the MS, MC, MM, or ML commands.

#### **Example:**

```
-K
get_line
scan_line+0B5
read_file+012C
main+03A

-K 2
get_line (000000001,0002AF44)
scan_line+0B5 (0000C252,FFFFFFFF)
read_file+012C (00000000,00000000)
main+03A (00000228,0000003)

-KA
get_line (000000001,0002AF44)
scan_line+0B5 (0000C252)
read_file+012C (00000000,00000000,0000A400,00000002)
main+03A (00000228,00000003)
```

## 3.14 M Command — Move Memory

#### Syntax:

M range address

## Description:

The move memory command copies a block of memory from one location to another. The command requires three operands. The first two operands specify the address range of the block to be copied. The third operand specifies the location to which the block is to be copied. The source and destination blocks may not overlap. If they do, the contents of the destination block will be unpredictable. If no segment selector is specified with the start address or the destination address, the segment pointed to by DS is assumed.

#### **Example:**

The command

-M 100 1FF 200

moves locations DS:100 through 1FF to locations DS:200 through 2FF.

The command

-M bufl buf1+20 buf2

copies the first 32 bytes of "buf1" to the beginning of "buf2".

## 3.15 MC, ML, MM, MS, and M? Commands — Select Memory Model

#### Syntax:

MC

ML

MM

MS

M?

## Description:

The select memory model commands specify the memory model of the program being debugged. 386 | DEBUG must know this information in order to correctly execute the K and KA (stack trace) commands, and in order to correctly use the \* indirection operator. The memory model of a program is determined at the time the program is compiled.

The MS command selects the small memory model (NEAR code and NEAR data pointers). The MC command selects the compact memory model (NEAR code and FAR data pointers). The MM command selects the medium memory model (FAR code and NEAR data pointers). The ML command selects the large memory model (FAR code and FAR data pointers). The default memory model assumed by 386 | DEBUG is small model.

The M? command displays the memory model currently assumed by 386 | DEBUG.

#### **Example:**

-MM Select medium model

-ML Select large model

-M?

Current memory model = LARGE

## 3.16 O Command — Output to Port

## Syntax:

O port value

## Description:

The output to port command outputs a data byte to an output port. The command takes two operands. The first operand is the number of the port to which the data byte is to be sent. The second operand is the data byte to write to the port.

Warning:

Misuse of this command can crash your system and require a reboot. When using the O command, always make sure that the port number and data byte are valid.

## Example:

To output a byte value of FF to port 100:

-0 100 FF

## 3.17 P and PQ Commands — Procedure Trace (Single Step Across CALL and INT Instructions)

## Syntax:

P [=address] [count]
PQ [=address] [count]

## Description:

The procedure trace commands trace the execution of a program one instruction at a time, while stepping over CALL instructions instead of

into the called procedure. These commands are useful when tracing execution through a procedure in which you are only interested in what that procedure does, not what any called procedures do.

The P and PQ commands also step to the next instruction after an INT instruction, rather than to two instructions after the INT as the T and TQ do.

The P command has a function similar to the T command except that it traces across CALL instructions instead of into the called procedure. Likewise, the PQ command traces quietly, as does the TQ command, except that it traces across CALL instructions. Section 3.26 describes the T and TQ commands.

## Example:

Trace three instructions at CS:EIP:

-PT 3

Trace ten instructions quietly:

-PQ A

## 3.18 PI and PL Commands — Display Paging Information

## Syntax:

PI address PL linaddr

## Description:

The display paging information commands show the contents of the page table entry for a specific address, and break apart the page table entry into some of its contents (the physical address of the memory page, the access rights for the page, and whether the page has been referenced by the program). See the Intel manual, 80386 Programmer's Reference Manual, for a complete description of page tables and page table entries.

The PI command takes a normal segment:offset address. If no segment is specified, the segment pointed to by DS is assumed.

The PL command takes a linear address. A linear address is formed by adding the offset within a segment to the base address of the segment.

## Example:

-pi 100
Linear address for 0014:00000100 = 00468100
Page table entry for linear address 00468000=003B4267
Physical base addr=003B4000 access=rd/wr usage=write privilege=user
-dl 14 14
#0014 Base=00468000 Limit=0029BFFF Flags=92 USE32

Page table entry for linear address 00468000=003B4267 Physical base addr=003B4000 access=rd/wr usage=write privilege=user

## 3.19 PROT Command — Assume Protected Mode

## Syntax:

PROT

-pl 468000

## Description:

The PROT command tells 386 | DEBUG to treat all program addresses as protected mode addresses (see section 2.9). 386 | DEBUG assumes protected mode addresses for protected mode programs, and real mode addresses for real mode programs.

When debugging mixed real and protected mode programs, you may occasionally want to look at protected mode code or data while the program is running in real mode, or vice versa. The PROT and REAL commands give you this ability.

## Example:

PROT

## 3.20 Q Command — Quit

## Syntax:

0

## Description:

The quit command is used to exit 386 | DEBUG. Use it to end a debugging session and to go back to DOS in order to run other programs.

## Example:

-Q

C> (DOS prompt)

## 3.21 R Command — Display or Change Registers

## Syntax:

R [register [value]]

## Description:

The register command displays and changes the contents of internal 80386 registers. The command has three different forms. If no operands are given with the R command, then all of the general purpose 80386 registers display on the screen. The register display looks like this:

```
-R
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00001FFE

DS=0014 SS=0014 ES=0014 GS=0014
CS:EIP=000C:0000200 EFLAGS=00000202 NV UP EI PL NZ NA PO NC
000C:0000200 B409 MOV AH,09
```

The first line shows the four data registers EAX, EBX, ECX, and EDX. The second line shows the four address registers ESI, EDI, EBP, and ESP. The third line shows five of the six segment registers DS, SS, ES, FS, and GS. The fourth line shows the address of the next instruction to be executed (CS:EIP), and the contents of the flag register, EFLAGS. EFLAGS is shown both as a hexadecimal value and with mnemonics for the flag bits of most

interest. The following mnemonics indicate if a particular flag bit is on or off:

TABLE 3-1
FLAG MNEMONICS

Bit Number	Flag	Off	On	
17	Virtual 8086 Mode	blank	VM	
11	Overflow	NV	OV	
10	Direction	UP	DN	
9	Interrupt Enable	DI	EI	
7	Sign	PL	NG	
6	Zero	NZ	ZR	
4	Auxiliary Carry	NA	AC	
2	Parity	PO	PE	
0	Carry	NC	CY	

The second form of the R command displays the contents of a single 80386 register. In this form, the command operand is the name of the register to be displayed. Valid register names are:

32-bit data registers: EAX, EBX, ECX, and EDX

16-bit data registers: AX, BX, CX, and DX

8-bit data registers: AL, BL, CL, DL, AH, BH, CH, and DH

32-bit address registers: ESI, EDI, EBP, and ESP 16-bit address registers: SI, DI, BP, and SP

Segment registers: CS, DS, SS, ES, FS, and GS

Instruction pointer: EIP and IP

Control registers: CR0, CR2, and CR3

Debug registers: DR0, DR1, DR2, DR3, DR6, and DR7

Test registers: TR6 and TR7

The third form of the R command changes the contents of an 80386 register. In this form of the R command, two operands are given. The first operand is the name of the register to be changed. The second operand is a value to be loaded into the register.

Note that if you gain control in 386 | DEBUG by typing CTRL/C while a protected mode program is running, 386 | DEBUG will not allow register values to be modified. Breakpoints and watchpoints can still be set as usual after a CTRL/C.

#### **Example:**

-R ESI ESI=00000000 -R ESI 123 -R ESI ESI=00000123

## 3.22 R67, R67D, R67S, and R87 Commands — Display Floating Point Coprocessor Registers

## Syntax:

R67 R67D R67S R87

## Description:

The R67 and R67D commands display in double precision form the floating point registers in a Weitek 1167, 3167, or 4167 floating point coprocessor.

The R67S command displays in single precision form the floating point registers in a Weitek 1167, 3167, or 4167 floating point coprocessor.

The R87 command displays floating point registers from an 80287, 80387, or compatible floating point coprocessor.

## Example:

```
-R87
ST(0) = 0.0000000000E + 00
                          0000 00000000000000000
                                                   Tag=11 (Empty)
ST(1)=0.000000000E+00
                          0000 0000000000000000
                                                   Tag=11 (Empty)
ST(2) = 0.0000000000E + 00
                          0000 00000000000000000
                                                   Tag=11 (Empty)
ST(3) = 0.0000000000E + 00
                          0000 00000000000000000
                                                   Tag=11 (Empty)
ST(4)=0.000000000E+00
                          0000 0000000000000000
                                                   Tag=11 (Empty)
```

Tag=11 (Empty)

Tag=11 (Empty)

```
ST(7)=2.5000000000E+04
                       400D C35000000003F3A0
                                             Tag=11 (Empty)
Control Word=037F Status Word=4124 Tag Word=FFFF
-R67
WD0 = 0.00000000E + 00
                    WD2 = 2.49989997E + 04
                    Hex=40D869BFFACCF4D3
WD4 = 1.60012805E - 09
                    Hex=3E1B7D7004C31321
WD6 = -4.79770300E + 03
                     Hex=C0B2BDB3F8129D59
WD8 = 7.85398163E - 01
                    Hex=3FE921FB54442D18
WD10=1.78390320E+04
                    Hex=40D16BC20C49BA5E
                    WD12=0.00000000E+00
                    WD14=0.00000000E+00
WD16=0.00000000E+00
                    WD18=0.00000000E+00
                    WD20=0.00000000E+00
                    WD22=0.00000000E+00
                    WD24=0.00000000E+00
                    WD26=0.00000000E+00
                    WD28=0.00000000E+00
                    WD30=0.00000000E+00
                    Mode=03 ExcMask=FF CCode=80 AccExc=20
-R67s
                                   WS1 = 0.000000E + 00
                                                     Hex=00000000
WSO = 0.000000E + 00
                   Hex=00000000
                                                      Hex=FACCF4D3
                                   WS3 = -5.320971E + 35
                   Hex=40D869BF
WS2 = 6.762908E + 00
                                                     Hex=04C31321
                                   WS5 = 4.586185E - 36
                   Hex=3E1B7D70
WS4 = 1.518457E - 01
                                                       Hex=F8129D59
                                    WS7 = -1.189479E + 34
                   Hex=C0B2BDB3
WS6 = -5.585657E + 00
                                                     Hex=54442D18
                                   WS9 = 3.370281E + 12
                   Hex=3FE921FB
WS8 =1.821350E+00
                                                     Hex=0C49BA5E
                                   WS11=1.554056E-31
                   Hex=40D16BC2
WS10=6.544404E+00
                                                     Hex=00000000
                                   WS13=0.000000E+00
                   Hex=00000000
WS12=0.000000E+00
                                                     Hex=00000000
                                   WS15=0.000000E+00
                   Hex=00000000
WS14=0.000000E+00
                                                     Hex=00000000
                                   WS17=0.000000E+00
                   Hex=00000000
WS16=0.000000E+00
                                                     Hex=00000000
                                   WS19=0.000000E+00
                   Hex=00000000
WS18=0.000000E+00
                                   WS21=0.000000E+00
                                                     Hex=00000000
                  Hex=00000000
WS20=0.000000E+00
                                                     Hex=00000000
                                   WS23=0.000000E+00
                   Hex=00000000
WS22=0,000000E+00
                                                     Hex=00000000
                                   WS25=0.000000E+00
                   Hex=00000000
WS24=0.000000E+00
                                                     Hex=00000000
                                   WS27=0.000000E+00
                   Hex=00000000
WS26=0.000000E+00
                                                     Hex=00000000
                                   WS29=0.000000E+00
                  Hex=00000000
WS28=0.000000E+00
                                   WS31=0.000000E+00
                                                     Hex=00000000
                  Hex=00000000
WS30=0.000000E+00
                 CCode=80 AccExc=20
Mode=03 ExcMask=FF
```

3FFE 869C00000008000D

3FFF 8000000000000000

ST(5) = 5.2581787109E - 01

ST(6)=1.0000000000E+00

## 3.23 REAL Command — Assume Real Mode Addressing

## Syntax:

REAL

## Description:

The REAL command tells 386 | DEBUG to treat all program addresses as real mode addresses (see section 2.9). 386 | DEBUG assumes protected mode addresses for protected mode programs, and real mode addresses for real mode programs.

When debugging mixed real and protected mode programs, you may occasionally want to look at protected mode code or data while the program is running in real mode, or vice versa. The PROT and REAL commands give you this ability.

#### **Example:**

REAL

## 3.24 RX Command — Display Extended Registers

## Syntax:

RX

## Description:

The display extended registers command displays all internal registers of the 80386 CPU. Like the R command, RX displays the general registers (EAX, EBX, etc.), the segment registers (DS, CS, etc.), and the flags. In addition, the RX command also displays the following special 80386 registers:

- The debug registers DRO, DR1, DR2, DR3, DR6, and DR7;
- The control registers CR0, CR2, and CR3;

- The test registers TR6 and TR7;
- The TR, GDTR, LDTR, and IDTR registers.

## Example:

```
-RX
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=000000000 EBP=00000000 ESP=00001FFE
DS=0014 SS=0014 ES=0014 FS=0014 GS=0014
CS:EIP=000C:00000200 NV UP EI PL NZ NA PO NC
CR0=00000001 CR2=00000000 CR3=0000F000
DR0=00021360 DR1=00000000 DR2=00000000 DR3=00000000
DR6=0000E00F DR7=00000003
TR6=00000000 TR7=00000000
GDTbase=00011D96 GDTlimit=002F IDTbase=00011EF4 IDTlimit=07FF
LDT=0028 TR=0000
```

## 3.25 S Command — Search Memory for String

## Syntax:

S range values

## Description:

The search command searches a block of memory for the occurrence of a string of data bytes. The first two command operands specify the address range of the memory block to be searched. The remaining operands specify the string of bytes for which memory is to be searched. The values may be given as separate bytes or as character strings. If no segment selector is given with the start address, the segment pointed to by DS is assumed.

The S command searches the entire specified range for each occurrence of the specified byte or string of bytes. Each time the specified values are encountered, the address of the first byte in the value string is printed out.

## Example:

```
-DB 0 1F
0014:00000000 B4 09 BA 10 00 00 00 CD-21 66 B8 00 4C CD 21 00 4.:...M!f8.LM!.
0014:00000000 48 65 6C 6F 20 77 6F-72 6C 64 21 21 21 21 21 Hello world!!!!!
```

-S 0 1F CD 21
0014:00000007
0014:0000000D
-S DS:0 1F 'HELLO'
-S DS:0 1F 'Hello'
0014:00000010
-S 14:0 1F '!'
0014:00000008
0014:0000000E
0014:0000001B
0014:0000001C
0014:0000001D
0014:0000001F

## 3.26 T and TQ Commands — Trace (Single-Step)

## Syntax:

```
T [=address] [count]
TQ [=address] [count]
```

## Description:

The trace commands trace the execution of a program one instruction at a time. In its simplest form, the T command takes no operands. It executes one 80386 instruction and then displays the 80386 registers on the screen. The registers display in the same form as the R command. The instruction addressed by instruction pointer CS:EIP is the instruction which is executed.

If a count is given as an operand to the T command, then this count specifies the number of instructions to be executed before stopping. The 80386 registers display after each instruction is executed. This form of the T command is useful for tracing a program without having to keep reentering the T command. The character \* can be used to specify an infinite count. If this value is given, instructions will be traced until a processor exception occurs or the program terminates.

The TQ command (trace quietly) is the same as the T command, except that the registers are only printed out after the specified number of instructions are traced or a processor exception occurs, instead of after every instruction traced. When 386 | DEBUG gains control after a TQ instruction, it prints out the number of instructions that were traced.

A second option to the T and TQ commands allows the start address for execution to be specified. Simply enter an equal sign (=) followed by the address, to begin execution. If no segment selector is specified for the start address, then the selector in the CS register will be used by default.

When the instruction to be traced is a software interrupt (INT) instruction, two instructions rather than one are actually traced, due to the 80386 architecture. The T instruction prints the message "Stepped two instructions to . . ." as a reminder after tracing over an INT instruction. The P and PQ commands can be used to trace only one instruction after an INT instruction.

## **Example:**

Trace five instructions at CS:EIP:

-T 5

Trace ten instructions at CS:100, only display the registers after the tenth instruction:

-TQ = 100 A

Trace instructions until a processor exception occurs or the program terminates:

-TQ \*

## 3.27 U, U16, U32 Commands — Unassemble (Disassemble) a Block of Memory

## Syntax:

U [range] U16 [range] U32 [range]

## Description:

The U, U16, and U32 commands disassemble a block of memory and display its contents as 80386 instructions. These commands take two optional operands, which specify the range of memory to be unassembled. If no operands are specified, 32 bytes will be unassembled, beginning one byte past the last byte unassembled the last time the command was executed. If only a start address is specified, 32 bytes will be unassembled, beginning at the start address. If both a start address and an end address are specified, the entire block will be unassembled. If no segment selector is specified for the start address to unassemble, then the selector in the CS register will be used by default.

For each instruction unassembled, 386 | DEBUG displays a line with the instruction's address, the hexadecimal values for the opcode bytes, and the assembly language statement for the instruction and its operands. Operand size is specified with the PTR operator where necessary, and segment overrides are indicated with the : operator.

Before unassembling the instruction at the start address, the offset of the instruction from the nearest previous symbol is displayed. If the unassemble command crosses into a new function while unassembling a block of instructions, the name of the new function is displayed.

In order to correctly unassemble a block of code, 386 | DEBUG must know whether the default operand size of the code segment descriptor is 16 bits or 32 bits.

If the U16 or U32 forms of the unassemble command are used, 386 | DEBUG assumes a default operand and address size based on the name of the command. U16 instructs 386 | DEBUG to assume a default operand and address width of 16 bits, while U32 causes it to assume a default width of 32 bits.

The U form of the unassemble command does not specify a default operand or address size, and 386 | DEBUG will infer one using the following logic:

If the program is running in real mode, a default size of 16 bits is assumed. This is a requirement of the 80386 architecture.

If the program is running in protected mode, 386IDEBUG uses the segment selector for the address being unassembled to make its size assumption. It looks up the selector in the appropriate descriptor table and extracts the operand size information directly from the table entry.

In most cases, the U form of the unassemble command suffices. The U16 and U32 commands are sometimes necessary when writing systems code, or when unassembling code in a data segment (for example, disassembling DOS code in the USE32 data segment that maps the low one megabyte of physical memory).

## **Example:**

Example unassembly for real mode:

```
-u MAIN MAIN+B
MAIN:
4E46:00000000 B84C4E MOV AX,4E4C
4E46:00000003 8ED8 MOV DS,AX
4E46:00000005 66BA1F000000 MOV EDX,0000001F
4E46:0000000B E81900 CALL 0027
```

## Example unassembly for protected mode:

```
-u 2e40 2e48

SCAN+056:

000C:00002E40 0400 ADD AL,00

000C:00002E42 00E8 ADD AL,CH

000C:00002E44 A2070000B4 MOV [B4000007],AL
```

## Example of forcing a 32-bit unassembly:

```
-u32 SCAN+6f SCAN+79

SCAN to 6F:

000C:00002A59 6650 PUSH AX

000C:00002A5B 8D1DE9000000 LEA EBX,[000000E9]

000C:00002A61 E8840B0000 CALL 000035EA
```

Example of unassembling some 16-bit code while in 32-bit protected mode (this code is part of the ROM BIOS):

```
-u16 34:f000b f0010
0034:000F000B 8BF0 MOV SI,AX
0034:000F000D B84000 MOV AX,0040
0034:000F0010 8ED8 MOV DS,AX
```

## 3.28 WC, WD, WE, WL, and WP Commands — Watchpoint Control

## Syntax:

```
WC watchpts
WC *
WD watchpts
WD *
WE watchpts
WE *
WL
WP address datalen [r|w]
```

## Description:

The watchpoint control commands set, clear, enable, disable, and list data watchpoints. Data watchpoints cause 386 | DEBUG to stop execution of the application immediately after execution of an instruction that reads or writes a specific memory location. When a data watchpoint occurs, 386 | DEBUG displays the memory location that is accessed, and the current register contents, and then prompts for a command. Data watchpoints can be useful for finding bugs which cause code or data locations in the program to be corrupted.

The WC command clears watchpoints. It takes as arguments either a list of watchpoint numbers, or the special character \*, which is a shorthand for all watchpoints. Once a watchpoint has been cleared, it will no longer be listed by the WL command and cannot be enabled with the WE command. If the watchpoint is needed again after being cleared, it must be reset with the WP command.

The WD and WE commands disable and enable watchpoints. They take as arguments either a list of watchpoint numbers, or the special character \*, which is a shorthand for all watchpoints. The WD command is used to temporarily disable watchpoints that are not desired to be active. The WE command is used to enable watchpoints that have been previously disabled with the WD command.

The WL command lists watchpoints that have been set with the WP command. Each watchpoint is identified by a number that is assigned by 386 | DEBUG. For each watchpoint, the WL command lists its watchpoint number, the address of the location the watchpoint is set on, and whether it is a write-only or a read/write watchpoint. In addition, if the watchpoint has been disabled, the WL command will print "\*\*\*disabled\*\*\*" following the watchpoint information.

The WP command sets watchpoints. The first command operand is the address of location to be watched. The second operand gives the data size of the expected access (one, two, or four bytes). If the wrong data size is used, the hardware will not detect an access to the memory location. The third operand is optional and specifies whether the watchpoint is to occur only on write accesses, or for any type of access (read or write). The character "w" is used to specify a write-only watchpoint, and the character "r" is used to specify a read/write watchpoint. If no third operand is given, a write-only watchpoint is set.

## Example:

```
-WP buf+21 1
-WP bufp 4 r
-WP errf 2 w
-WL
0 BUF+021 datalen=1 write only
1 BUFP datalen=4 read/write
2 ERRF datalen=2 write only
-WD 1
-WL
0 BUF+021 datalen=1 write only
1 BUFP datalen=4 read/write ***disabled***
2 ERRF datalen=2 write only
-WE *
-WC 0 2
-WL
1 BUFP datalen=4 read/write
```

# 3.29 X Command — Display Symbol Table

## Syntax:

X [symbol]

# Description:

The display symbol table command displays the public symbols and their values for the program being debugged. If no command operand is given with the X command, the entire symbol table is displayed. If a string value is given as a command operand, all symbols whose names begin with the specified string are displayed. CTRL/S can be used to stop scrolling, and CTRL/C to return to the prompt.

## Example:

-X ma
MAIN = 0000000A
MAKE\_PTR = 0000032C
MARK\_POS = 000045B0
-X mak
MAKE\_PTR = 0000032C

# 3.30 XR Command — Relocate Symbol Table

## Syntax:

XR old\_sel new\_sel

# Description:

Use the XR command to inform 386 | DEBUG that the symbol table applies to code and data in segments other than the default code and data segments of 000Ch and 0014h. This command allows symbolic debugging of separate add-in drivers or overlays that are loaded by the main program at runtime.

Use the -SYMFILE switch when starting 386 | DEBUG to load the symbol table for the add-in driver rather than the main program. After the main

program has loaded the driver, use the XR command to relocate the symbol table from segments 000Ch and 0014h to the segments used for the add-in driver.

Please see section 2.5 for more information on debugging with an alternate symbol table.

## Example:

-XR c 7c

# 3.31 XS Command — Display Segment Names

## Syntax:

XS

# Description:

The display segment names command displays the name, the segment selector value (the values that are loaded into segment registers to address the segments), and the size in bytes of each of the program's segments. For protected mode programs, the linker constructs a single large program segment, and assigns the segment name FLAT\_CODE to selector 000C, and the name FLAT\_DATA to selector 0014. Therefore, the XS command will always show the same information for protected mode programs. For real mode programs, the segments and group names are shown as they are defined in the source code, and the segment values are the paragraph base address of the segment or group.

### Example:

For a protected mode program:

-XS				
FLAT CODE				0001A800
FLAT_DATA	0014	Size	=	0001A800

# For a real mode program:

-XS				
BSS	2E76	Size	=	00000200
CODE	2350	Size	=	0000B260
DATA	2E96	Size	=	000039E4
DGROUP	2E76	Size	=	00003BE4

# 3.32 XW Command — Locate Symbol

### Syntax:

XW address

# Description:

The locate symbol command finds the nearest previous symbol to a specific address in a program. The single command operand is the program address to be looked up in the symbol table. The nearest previous symbol and the offset from the symbol's address to the specified address is displayed. If the specified address is exactly equal to the symbol's address, then the symbol is displayed with no offset.

## Example:

-XW 400 CHK\_ERR+01C

-XW dgroup:30 DATBUF

# 3.33 ? Command — On Line Help

#### Syntax:

? [character]

## Description:

The ? command obtains online help with 386 | DEBUG command formats. The help screen lists each command, shows the operands to each command, and gives a one-line description of the command.

If a single character prefix is given, only the commands that start with that character are listed.

## Example:

```
-? T
T [=address] [count] Trace (single-step)
TQ [=address] [count] Trace quietly
```



# 386 | DEBUG Command Line

386DEBUG [debugswitches] programname arguments

debugswitches Optional; override the default operation of 386 | DEBUG

Name of .EXP, .EXE, or .REX file to be debugged programname

Parameters passed to program being debugged arguments

#### 386 | DEBUG Commands:

Clear breakpoints BC breakpoints | \* Disable breakpoints BD breakpoints | \* Enable breakpoints BE breakpoints | \*

List all breakpoints BL Set breakpoint

BP address Compare memory C range address

Redirect I/O to serial channel #1 COM1

Redirect I/O to serial channel #2 COM2

Return I/O to console CON

Dump memory D [range]

Dump memory in ASCII DA [range] Dump memory as hex bytes DB [range]

Dump memory as hex double words DD [range]

Dump memory as 4 byte floats DF [range]

Dump memory as hex words

DW [range] Dump GDT

DG [range] Dump IDT DI [range] Dump LDT DL [range]

Dump memory as 8 byte floats DQ [range]

Dump memory as 4 byte floats DS [range]

Dump memory as 10 byte floats DT [range]

Use LARGE memory model Use MEDIUM memory model Use SMALL memory model Show current memory model Output to port Trace (skipping CALL instrs.) Show paging info for address Show paging info for linear addr Assume protected mode addresses Trace quietly (skipping CALL instrs.) Quit Display or change registers Display 1167/3167/4167 registers Display 1167/3167/4167 registers Display 80287/80387 registers Assume real mode addresses Display extended registers Search memory for string Trace (single-step) Trace quietly Unassemble memory
Trace (single-step)

WD watchpoints   *	Disable watchpoints
WE watchpoints   *	Enable watchpoints
WL	List all watchpoints
WP address datalen [r w]	Set watchpoint
X [symbol]	Display symbol table
XR old sel new sel	Relocate symbols to another selector
XS	Display segment names
XW address	Locate symbol at address
? [char]	Display debugger commands

# **Expression Formats:**

Values	12A	Hexadecimal number
	'ABC'	String
	AL	Contents of a register
	F5+AA	Addition
	FF-12	Subtraction
Addresses	12A	Offset
	main	Symbol
	CS:145	Segment register override
	8:145	Selector override
	ESP	Indirect through a register
	SS:ESP	Indirect through a register with
		segment override
	buf+10	Addition
	EBP-10	Subtraction
Range	buf buf+FF	Simple range
	CS:100 1FF	Range with segment override
	14:100 1FF	Range with selector override



# **Error Messages**

This appendix lists 386 | DEBUG errors, their causes, and how to correct them. The error messages are divided into two sections. The first section lists errors that can occur when 386 | DEBUG attempts to load your program before the first command prompt is displayed. The second section lists errors that can occur when commands are entered. The error messages are listed in alphabetical order within each section.

# **B.1** Program Load Errors

Error:

? Can't open executable file: filename

Cause:

An attempt has been made to debug a nonexistent .EXP,

.EXE, or .REX file.

Solution:

1) Correct the spelling of the name of the .EXP file, or

2) Give the correct directory name in which the .EXP file

resides, or

3) Create the .EXP file with 386 LINK.

Error:

? Error reading file: filename

Cause:

The .EXP file is damaged.

Solution:

Rebuild the .EXP file, making sure there are no link

errors.

Error:

Cause:

Error:

Cause:

Solution:

An unknown error occurred when MS-DOS attempted to load a real mode .EXE file. Solution: Reboot your system. Error: ? Missing file name Cause: An attempt has been made to run 386 | DEBUG without specifying a program file to be debugged. Solution: Give the name of an .EXP file containing a program to be debugged. ? Not an executable file: filename Error: Cause: The file is not an .EXP, .EXE, or .REX format file.

? Insufficient memory to load program

1) Shrink your program, or 2) Purchase more memory, or

? Load error

Your program is larger than the available memory.

3) Remove memory resident programs from your system.

#### Command Errors B.2

Error:

? Bad command operand

Cause:

An operand to a command is ill-formed. For example:

1G

(Illegal hexadecimal digit)

1+

(Missing right operand to +)

ABC

(Missing ending quote)

Solution:

Correct the operand, and re-enter the command.

Error:

? Can't alter registers after CTRL/C in protected mode

Cause:

An attempt has been made to modify one of the 80386 registers after getting control with a CTRL/C when the program being debugged was executing in protected mode. This is not permitted.

Error:

? Extra characters at end of line

Cause:

After the last operand of a command, more characters were found on the line.

Solution:

Remove the extra characters from the line, and re-enter the command.

Error:

? Invalid address: segment:offset

Cause:

An address which either has an invalid segment selector value or an offset beyond the segment limit has been specified.

? Missing required command operand Error: At least one operand required by the command was not Cause: given. Solution: Use the help (?) command to determine the required command format. ? Only count bytes could actually be entered Error: Cause: Fewer bytes than requested were actually written into memory by one of the enter commands (E, EB, EW, ED), because the end of the segment was reached. Error: ? Only count bytes were actually filled Cause: Fewer bytes than requested were actually written into memory by the fill (F) command, because the end of the segment was reached. Error: ? Segment selector: value not readable The specified segment cannot be read. Cause: Solution: Use another segment that maps the same memory to read the data. Error: ? Segment selector: value not writable Cause: The specified segment cannot be written. Solution: Use another segment that maps the same memory to write the data.

Error:

? Unknown command

Cause:

An illegal command name was given to 386 | DEBUG.

Solution:

Correct the spelling of the command name. Please see

Appendix A for a list of valid commands.

Error:

? Value out of range

Cause:

A value given as a command operand exceeded the largest

value allowed by the command.



# Index

386debug command, 3 386DEBUG.EXE file, 3 switches configured into, 8 80287 floating point coprocessor, 52 80387 floating point coprocessor, 52

\* (asterisk)
for comment lines, 22
as indirection operator, 17-18
as indirection operator, in command operands,
11
as indirection operator, in expressions, 13
in instruction traces, 56
in watchpoint commands, 60
: (colon)

in addresses, 11 as segment override operator, 13, 15 segment override operator, 17

= (equal sign), for starting addresses, 41

> (greater-than character), for redirection, 22

< (less-than character), for redirection, 22

(minus sign)
 in command line switches, 5
 in command operands, 11
 in expressions, 13
 as subtraction operator, 16
["(",")"] (parentheses)
 in command operands, 11

in expressions, 13
+ (plus sign)
as addition operator, 16
in command operands, 11
in expressions, 13

? command, 65 ' (single quotation mark), 11 A

absolute numbers, 11, 12
addition operator (+), 16
addresses, 14-15
ranges for, 15-16
REAL command for, 54
ASCII format, DA command to display memory
contents in, 33
assembly language programs, PUBLIC directive
in, 11
asterisk (\*)
for comment lines, 22
as indirection operator, 17-18

as indirection operator, in command operands,
11
as indirection operator, in expressions, 13
in instruction traces, 56
in watchpoint commands, 60

B

-BAUD switch, 5-6, 21, 32
BC command, 23, 29-31
BD command, 29-31
BE command, 29-31
BL command, 29-31
blocks of memory, commands for unassembly of, 57-59
BP command, 23, 29-31

BP command, 23, 29-31 breakpoints, 23-24 commands for, 29-31 G command to set, 41

С	to display task state segment, 38-39 to enter or change memory, 39-40
	error messages for, 73-75
C (language)	to fill memory, 40-41
public symbols in, 11-12	formats for, 9
stack traces in, 44	go, 41-42
calculator, H command for, 42	for input from ports, 43
CALL instruction, traces and 47-48	for instruction traces, 56-57
case sensitivity switches, 6-7, 12	to locate symbols, 64
C command, 31-32	to move memory, 45-46
CFIG386 utility program, 8	not used after program termination, 25-26
colon (:)	
in addresses, 11	for on-line help, 65
as segment override operator, 13, 15	to output to port, 47
segment override operator, 17	for procedure traces, 47-48
COM1 command, 32-33	for real mode addressing, 54
-COM1 switch, 5, 20-21	redirection of, 22
COM2 command, 32-33	to relocate symbol tables, 62-63
-COM2 switch, 5, 21	to search memory for strings, 55-56
command formats	to select memory models, 46-47
addresses in, 14-15	for stack traces, 43-45
expression operators in, 16-18	to unassemble blocks of memory, 57-59
expressions in, 12-14	for watchpoint control, 60-61
operands in, 10-11	communications ports
ranges in, 15-16	-COM1 and -COM2 switches for, 5-6
symbols in, 11-12	commands for redirection to, 32-33
command line, 67-69	redirection to, 20-21
syntax of, 3-4	COMPACT memory model
command line switches, 4-5	indirection operator (*) and, 17-18
	-COMPACT switch, 7
case sensitivity, 6-7	indirection operator (*) and, 17
default pointer size, 7	CON command, 32-33
I/O redirection, 5-6	CTRL/C command, 20, 21, 26
for 386 I DOS-Extender, 8-9	R command not used after, 52
commands, 67-69	to stop I/O redirection, 32
to assume protected mode, 49	control of program execution
for breakpoint control, 29-31	breakpoints and instruction tracing for, 23-24
to display extended registers, 54-55	CTRL/C command for, 26
to display floating point coprocessor registers,	data watchpoints for, 24
52-53	processor exceptions and, 24-25
to display memory, 33-34	program termination, 25-26
to display memory as floating point numbers, 35-36	CTRL/S command, 20, 62
to display or change registers, 50-52	coprocessor registers, commands to display, 52-53
to display page information, 48-49	CR0 register, 54
to display segment names, 63-64	CR2 register, 54
to display symbol tables, 62	CR3 register, 54
to display system tables, 36-38	CS register, 14

D
DA command, 33-34
data watchpoints, 24
commands for, 60-61
DB command, 33-34
D command, 33-34
DD command, 33-34
defaults
command line switches to change, 4-5
for pointer size, 7
DF command, 35-36
DG command, 36-38
DI command, 36-38
disassembly of blocks of memory, commands for,
57-59
disk files, I/O redirection to, 21-22
displays
controlling, 20
of extended registers, 54-55
of floating point coprocessor registers,
commands for, 52-53
of memory contents, commands for, 33-34
of memory contents, as floating point numbers,
commands for, 35-36
of page information, commands for, 48-49
redirection to communications ports of, 20-21
redirection to disk files of, 21-22
of registers, command for, 50-52
of segment names, command for, 63-64
of symbol tables, command for, 62
of system tables, commands for, 36-38
of task state segment, commands for, 38-39
DL command, 36-38 DOSX environment variable, 8
DQ command, 35-36
DR0 register, 54
DR1 register, 54
DR2 register, 54
DR3 register, 54
DR6 register, 54
DR7 register, 54

DTSS16 command, 38-39 DTSS32 command, 38-39 DTSS command, 38-39 DW command, 33-34

#### E

EAX register, 50 EB command, 39-40 EBP register, 50 EBX register, 50 E command, 39-40 ECX register, 50 ED command, 39-40 EDI register, 50 EDX register, 50 EFLAGS register, 50-51 80287 floating point coprocessor, 52 80387 floating point coprocessor, 52 environment variables, DOSX, 8 equal sign (=), for starting addressed, 41 error messages for command errors, 73-75 for program load errors, 71-72 ESI register, 50 ESP register, 50 ES register, 50 EW command, 39-40 exceptions, 24-25 execution, control of breakpoints and instruction tracing for, 23-24 CTRL/C command for, 26 data watchpoints for, 24 processor exceptions for, 24-25 program termination, 25-26 .EXE file extension, 4 .EXE file format, 4 for real mode programs, 26 symbol tables in, 9, 11 -SYMFILE switch to load symbol table from, 8 exiting, quit command for, 50 .EXP file format, 4 alternate symbol tables for, 19-20 for protected mode programs, 26

symbol tables in, 9, 11

drivers, symbol tables for, 19

DS command, 35-36

DT command, 35-36

DS register, 14, 50

-MEDIUM switch, 7 indirection operator (\*) and, 17 memory addresses for, 14-15 C command to compare, 31-32 comands to display as floating point numbers, 35-36 commands to display, 33-34 commands to enter or change, 39-40 command to fill, 40-41 command to move, 45-46 command to search for strings in, 55-56 commands to unassemble blocks of, 57-59 memory models commands to select, 46-47 indirection operator (\*) and, 17-18 messages for command errors, 73-75 for program load errors, 71-72 MetaWare High C (language), 44 Microway NDP C-386 (language), 44 Microway NDP FORTRAN-386 (language), 44 minus sign (-) in command line switches, 5 in command operands, 11 in expressions, 13 as subtraction operator, 16 ML command, 46-47 MM command, 46-47 MODE command (MS-DOS), 5, 21, 32 -BAUD switch to override, 6 MS command, 46-47 MS-DOS operating system, CTRL/C command to return to, 26 N

NEAR memory model, 46-47 NEAR pointers, 18

#### 0

O command, 47
offset
in addresses, 14
for symbol values in real mode programs, 12
-ONECASE switch, 6
on-line help command, 65
operands, 10-11
operators
addition and subtraction, 16-17
indirection, 17-18
order of precedence of, 13
segment override, 17
outputs
displays, controlling, 20
to ports, command for, 47

#### P

pages, commands to display information on, 48-49 parameters, for communications ports, 5-6, 21, 32 parentheses ["(",")"] in command operands, 11 in expressions, 13 Pascal (language), 44 P command, 47-48 not used after program termination, 25-26 permanent breakpoints, 23 PI command, 48-49 PL command, 48-49 plus sign (+) as addition operator, 16 in command operands, 11 in expressions, 13 pointers default pointer size switches for, 7 indirection operator (\*) for, 17-18 ports -COM1 and -COM2 switches for, 5-6 commands for redirection to, 32-33 commands to output to, 47 I command for input from, 43 redirection to, 20-21 PO command, 47-48

procedure call stacks, traces for, 43 procedure traces, commands for, 47-48 processor exceptions, 24-25 Professional Pascal (language), 44 program load errors, messags for, 71-72 programs breakpoints and instruction tracing in, 23-24 termination of, 25-26 PROT command, 49 protected mode programs addresses in, 14 CTRL/C command used in, 26 debugging, 26-27 default pointer size switches for, 7 PROT command for, 49 unassembly of blocks of memory in, 59 PUBLIC directive (assembly language), 11 public symbols, 12 command to display, 62 in symbol tables, 9

#### Q

Q command, 50 ? command, 65 quit command, 50

R67 command, 52-53

R67D command, 52-53

R67S command, 52-53

#### R

R87 command, 52-53
ranges, 15-16
error message regarding, 75
R command, 50-52
REAL command, 54
real mode programs
CTRL/C command used in, 26
debugging, 26-27
REAL command for, 54
segment selector in, 14
symbol values in, 12
unassembly of blocks of memory in, 58

redirection commands for, 32-33 to communications ports, 20-21 to disk files, 21-22 registers command to display or change, 50-52 after CTRL/C command, 26 extended, commands to display, 54-55 floating point coprocessor, commands to display, 52-53 in 386 | DEBUG commands, 11 used in expressions, 15 .REX file format, 4 run386 command, 3 RUN386.EXE file, 3 switches configured into, 8 RX command, 54-55

#### S

S command, 55-56 segment names, command to display, 63-64 segment override operator (:), 13, 17 segment selectors, 26 single quotation mark ('), 11 -SMALL switch, 7 indirection operator (\*) and, 17 SS register, 50 stack trace commands, 43-45 default pointer size switches for, 7 starting 386 | DEBUG, 9-10 strings, command to search memory for, 55-56 subtraction operator (-), 16 switches, 4-5 case sensitivity, 6-7 default pointer size, 7 I/O redirection, 5-6 for 386 | DOS-Extender, 8-9 symbols, 11-12 command to locate, 64 -SYMBOLS switch, 9 symbol tables, 9, 11-12 alternate, loading, 19-20 command to display, 62 command to relocate, 62-63 -SYMFILE switch to load, 8

-SYMFILE switch, 8, 19-20, 62 syntax of command line, 3-4 system tables, commands to display, 36-38

SYMDEB utility, 1

task register (TR), 38

V

variables, DOSX environment variable, 8

T

task state segments, commands to display, 38-39

T command, 23, 56-57 not used after program termination, 25-26 temporary breakpoints, 23

G command to set, 41 termination of programs, 25-26 386 | DEBUG, 1

command line in, 67-69 controlling display output from, 20 quit command to exit, 50 starting, 9-10 386debug command, 3

386DEBUG.EXE file, 3 switches configured into, 8 386 DOS-Extender, 1, 3 switches for, 8-9 386 | SRCBug, 1

TR6 register, 55 TR7 register, 55 traces commands for, 56-57 instruction traces, 23-24 procedure traces, 47-48

TQ command, 56-57

stack traces, 43-45 TR register, 55 -TWOCASE switch, 6, 12

U

U16 command, 57-59 U32 command, 57-59 U command, 23, 57-59 unassembly of blocks of memory, commands for, 57-59

W

watchpoints, 24 commands for, 60-61 Watcom C-386 (language), 44 WC command, 60-61 WD command, 60-61 WE command, 60-61 WL command, 60-61

WP command, 24, 60-61

X

X command, 12, 62 XR command, 19, 20, 62-63 XS command, 63-64 XW command, 64



>++>+O++++

The people who wrote, edited, revised, reviewed, indexed, formatted, polished, and printed this manual were:

John Benfatto, Alan Convis, Noel Doherty, Lorraine Doyle, Bryant Durrell, Diego Escobar, Nan Fritz, Mike Hiller, Elliot Linzer, Bob Moote, Kim Norgren, Richard Smith and Hal Wadleigh.





